Actions

Each pattern in a rule has a corresponding action, which can be any arbitrary C statement. The pattern ends at the first non-escaped whitespace character; the remainder of the line is its action. If the action is empty, then when the pattern is matched the input token is simply discarded. For example, here is the specification for a program which deletes all occurrences of "zap me" from its input:

```
%%
"zap me"
```

(It will copy all other characters in the input to the output since they will be matched by the default rule.)

Here is a program which compresses multiple blanks and tabs down to a single blank, and throws away whitespace found at the end of a line:

```
%%
[ \t]+        putchar( ' ' );
[ \t]+$       /* ignore this token */
```

If the action contains a '{', then the action spans till the balancing '}' is found, and the action may cross multiple lines. `flex` knows about C strings and comments and won't be fooled by braces found within them, but also allows actions to begin with `` `%{' `` and will consider the action to be all the text up to the next `` `%}' `` (regardless of ordinary braces inside the action).

An action consisting solely of a vertical bar ('|') means "same as the action for the next rule." See below for an illustration.

Actions can include arbitrary C code, including `return` statements to return a value to whatever routine called `` `yylex()' ``. Each time `` `yylex()' `` is called it continues processing tokens from where it last left off until it either reaches the end of the file or executes a return.

Actions are free to modify `yytext` except for lengthening it (adding characters to its end--these will overwrite later characters in the input stream). This however does not apply when using `` `%array' `` (see above); in that case, `yytext` may be freely modified in any way.

Actions are free to modify `yyleng` except they should not do so if the action also includes use of `` `yymore()' `` (see below).

There are a number of special directives which can be included within an action:

- `` `ECHO' `` copies yytext to the scanner's output.
- `BEGIN` followed by the name of a start condition places the scanner in the corresponding start condition (see below).
- `REJECT` directs the scanner to proceed on to the "second best" rule which matched the input (or a prefix of the input). The rule is chosen as described above in "How the Input is Matched", and `yytext` and `yyleng` set up appropriately. It may either be one which matched as much text as the originally chosen rule but came later in the `flex` input file, or one which matched less text. For example, the following will both count the words in the input and call the routine special() whenever "frob" is seen:

```
        int word_count = 0;
%%
```

```
frob        special(); REJECT;
[^ \t\n]+   ++word_count;
```

Without the `REJECT`, any "frob"'s in the input would not be counted as words, since the scanner normally executes only one action per token. Multiple `REJECT's` are allowed, each one finding the next best choice to the currently active rule. For example, when the following scanner scans the token "abcd", it will write "abcdabcaba" to the output:

```
%%
a        |
ab       |
abc      |
abcd     ECHO; REJECT;
.|\n     /* eat up any unmatched character */
```

(The first three rules share the fourth's action since they use the special '|' action.) `REJECT` is a particularly expensive feature in terms of scanner performance; if it is used in *any* of the scanner's actions it will slow down *all* of the scanner's matching. Furthermore, `REJECT` cannot be used with the `` `-Cf' `` or `` `-CF' `` options (see below). Note also that unlike the other special actions, `REJECT` is a *branch*; code immediately following it in the action will *not* be executed.

- `` `yymore()' `` tells the scanner that the next time it matches a rule, the corresponding token should be *appended* onto the current value of `yytext` rather than replacing it. For example, given the input "mega-kludge" the following will write "mega-mega-kludge" to the output:

  ```
  %%
  mega-    ECHO; yymore();
  kludge   ECHO;
  ```

  First "mega-" is matched and echoed to the output. Then "kludge" is matched, but the previous "mega-" is still hanging around at the beginning of `yytext` so the `` `ECHO' `` for the "kludge" rule will actually write "mega-kludge".

Two notes regarding use of `` `yymore()' ``. First, `` `yymore()' `` depends on the value of `yyleng` correctly reflecting the size of the current token, so you must not modify `yyleng` if you are using `` `yymore()' ``. Second, the presence of `` `yymore()' `` in the scanner's action entails a minor performance penalty in the scanner's matching speed.

- `` `yyless(n)' `` returns all but the first *n* characters of the current token back to the input stream, where they will be rescanned when the scanner looks for the next match. `yytext` and `yyleng` are adjusted appropriately (e.g., `yyleng` will now be equal to *n*). For example, on the input "foobar" the following will write out "foobarbar":

  ```
  %%
  foobar    ECHO; yyless(3);
  [a-z]+    ECHO;
  ```

  An argument of 0 to `yyless` will cause the entire current input string to be scanned again. Unless you've changed how the scanner will subsequently process its input (using `BEGIN`, for example), this will result in an endless loop. Note that `yyless` is a macro and can only be used in the flex input file, not from other source files.
- `` `unput(c)' `` puts the character `c` back onto the input stream. It will be the next character scanned. The following action will take the current token and cause it to be rescanned enclosed in parentheses.

  ```
  {
  int i;
  /* Copy yytext because unput() trashes yytext */
  char *yycopy = strdup( yytext );
  unput( ')' );
  for ( i = yyleng - 1; i >= 0; --i )
      unput( yycopy[i] );
  unput( '(' );
  ```

```
free( yycopy );
}
```

Note that since each `unput()' puts the given character back at the *beginning* of the input stream, pushing back strings must be done back-to-front. An important potential problem when using `unput()' is that if you are using `%pointer' (the default), a call to `unput()' *destroys* the contents of yytext, starting with its rightmost character and devouring one character to the left with each call. If you need the value of yytext preserved after a call to `unput()' (as in the above example), you must either first copy it elsewhere, or build your scanner using `%array' instead (see How The Input Is Matched). Finally, note that you cannot put back EOF to attempt to mark the input stream with an end-of-file.

- `input()' reads the next character from the input stream. For example, the following is one way to eat up C comments:

```
%%
"/*"        {
            register int c;

            for ( ; ; )
                {
                while ( (c = input()) != '*' &&
                        c != EOF )
                    ;    /* eat up text of comment */

                if ( c == '*' )
                    {
                    while ( (c = input()) == '*' )
                        ;
                    if ( c == '/' )
                        break;    /* found the end */
                    }

                if ( c == EOF )
                    {
                    error( "EOF in comment" );
                    break;
                    }
                }
            }
```

(Note that if the scanner is compiled using `C++', then `input()' is instead referred to as `yyinput()', in order to avoid a name clash with the `C++' stream by the name of input.)

- YY_FLUSH_BUFFER flushes the scanner's internal buffer so that the next time the scanner attempts to match a token, it will first refill the buffer using YY_INPUT (see The Generated Scanner, below). This action is a special case of the more general `yy_flush_buffer()' function, described below in the section Multiple Input Buffers.
- `yyterminate()' can be used in lieu of a return statement in an action. It terminates the scanner and returns a 0 to the scanner's caller, indicating "all done". By default, `yyterminate()' is also called when an end-of-file is encountered. It is a macro and may be redefined.

Go to the first, previous, next, last section, table of contents.